Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems

R. O. Kirk*, G. R. Mudalige*, I. Z. Reguly†, S. A. Wright*, M. J. Martineau‡, S. A. Jarvis*
* Department of Computer Science, University of Warwick, Coventry, UK
{R.Kirk, G.Mudalige, Steven.Wright, S.A.Jarvis}@warwick.ac.uk
† Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary
reguly.istvan@itk.ppke.hu
‡ Department of Computer Science, Merchant Venturers Building, Woodland Road, Clifton, Bristol, UK
m.martineau@bristol.ac.uk

***Abstract*—Modernizing production-grade, often legacy applications to take advantage of modern multi-core and many-core architectures can be a difficult and costly undertaking. This is especially true currently, as it is unclear which architectures will dominate future systems. The complexity of these codes can mean that parallelisation for a given architecture requires significant re-engineering. One way to assess the benefit of such an exercise would be to use mini-applications that are representative of the legacy programs.**

In this paper, we investigate different implementations of TeaLeaf, a mini-application from the Mantevo suite that solves the linear heat conduction equation. TeaLeaf has been ported to use many parallel programming models, including OpenMP, CUDA and MPI among others. It has also been re-engineered to use the OPS embedded DSL and template libraries Kokkos and RAJA. We use these different implementations to assess the performance portability of each technique on modern multi-core systems.

While manually parallelising the application targeting and optimizing for each platform gives the best performance, this has the obvious disadvantage that it requires the creation of different versions for each and every platform of interest. Frameworks such as OPS, Kokkos and RAJA can produce executables of the program automatically that achieve comparable portability. Based on a recently developed performance portability metric, our results show that OPS and RAJA achieve an application performance portability score of 71% and 77% respectively for this application.

***Keywords*—Mini-apps, OPS, RAJA, Kokkos, Performance Portability**

I. Introduction

Modernizing production-grade, often legacy applications to take advantage of modern multi-core and many-core architectures can be a difficult and costly undertaking. Often, these applications have been developed over decades and consist of code bases with thousands or even millions of lines of code. Adapting to new systems may require major re-engineering, since languages, parallel programming models and optimisations vary widely between different platforms. At the same time, there is considerable uncertainty about which platforms to target; it is not apparent which parallel programming approach is likely to "win" in the long term. Clearly, manually porting large code-bases to use various different programming models and languages, and then maintaining these different versions, is infeasible.

One common strategy is to use small representative applications to test and evaluate new technologies, programming models, frameworks and optimisations. The use of such programs, called proxy or mini-applications, is not new. The idea can be traced to the development of small benchmark codes such as LINPACK [1] and the NAS Parallel Benchmarks [2]. More recent efforts include the Mantevo [3] and UK Mini-Application Consortium [4] suites. Due to their small size, mini-apps are much more manageable than production applications and can feasibly be re-written in different programming languages and with specific optimisations. They are also unrestricted and/or devoid of any commercially sensitive code, allowing them to be readily distributed to many parties and sites.

In this paper, we explore the performance of one such mini-app called TeaLeaf, recently developed as a proxy for algorithms of interest at the UK AWE plc. TeaLeaf implements a set of linear equations which form a sparse, structured mesh and use a five point stencil and cell-centred temperatures to calculate the conduction coefficient [5]. It has been parallelised using a variety of different programming models and language extensions, including OpenMP, MPI, CUDA and OpenACC. It also has been implemented using the OPS embedded domain specific language [6], and the C++ template libraries Kokkos [7] and RAJA [8]. Many of these programming frameworks allow for compilation and execution on multiple different systems and architectures.

Specifically, this paper makes the following contributions:

- First, we compare the performance of different implementations of TeaLeaf, including how manually parallelised and optimised versions compare to those using the frameworks OPS, Kokkos and RAJA;
- Second, we analyse the performance of TeaLeaf on a number of current multi-core systems including Intel's Xeon Phi Knights Landing (KNL) processor and NVIDIA's Tesla P100 GPU.

As part of this work, we examine the idea of performance portability – a measure of the performance gained by a single application across a range of different systems. An

application is said to be highly performance portable if it achieves the best execution possible (or close to best) on each platform it is tested on. We use a recently developed metric for performance portability in analysing the achieved performance of TeaLeaf developed with the above of programming models and frameworks [9].

The rest of the paper is organised as follows: in Section II, we discuss the background of mini-applications and briefly detail the development of TeaLeaf; Section III discusses the different implementations of TeaLeaf to achieve parallelism through different techniques; in Sections IV and V, we discuss the performance of the many versions of TeaLeaf and the resulting performance portability on the systems of interest; finally, Section VI, concludes the paper.

II. Background

Improving the performance of large-scale, production applications is a significant undertaking. Often, these applications have been developed over decades, by multiple teams, using several third party libraries and consist of code bases with thousands or even millions of lines of code. However, in many cases, the performance is dominated by a few units within the application. As such, a representative program, often smaller in size, can be created to act as a proxy of the original code. A key benefit of these representative applications is that they can be modified and deployed on a range of systems quickly, implemented with multiple parallelisation models and optimised using a wide range of techniques [3].

Notable efforts in developing and using mini-apps include the NAS Parallel Benchmarks in the late 1980s [2], the ASCI applications in the 1990s [10], and more recently the Mantevo [3] and UKMAC [4] benchmark suites. Mini-apps have been developed to represent production applications from a wide range of scientific and engineering areas, including CFD [2], [11], [12], particle transport [13], hydrodynamics [14], [15] and machine learning [16], to name just a few.

In this paper, we focus on the heat conduction solver mini-app TeaLeaf [17], part of the Mantevo and UKMAC suites. Martineau et al. [5], [18], [19] discuss several variants of TeaLeaf that have been parallelised using a number of programming models. Further, they compare different solvers within TeaLeaf: Conjugate Gradient (CG), Chebyshev and Chebyshev polynomially preconditioned CG (PPCG), on three different Intel Xeon processors, an IBM Power8 processor, an NVIDIA Tesla K20x GPU and an Intel Knights Corner accelerator card [5], [18], [19]. Recently, TeaLeaf was re-engineered to use the OPS [6] embedded domain specific language, and the Kokkos [7] and RAJA [8] C++ template libraries.

III. Parallelising TeaLeaf

TeaLeaf is one of 15 mini-applications within the Mantevo suite [3]. The reference version, and a number of versions capable of executing in parallel using MPI and OpenMP, are written in Fortran. In order to make use of other parallel programming models, the application has also been converted to C/C++. In this section we detail the different versions of TeaLeaf used in our study. We first describe the original reference application and a number of versions ported manually to make use of various parallel programming models. Secondly we detail the version parallelised using OPS. Finally, we describe versions parallelised by the C++ template libraries, Kokkos and RAJA.

A. Reference Implementation and Manual Parallelisations

The initial reference version of TeaLeaf employs both OpenMP and MPI to allow parallelisation on both shared and distributed memory systems. Subsequently, it has been manually ported to use other parallel programming models. TeaLeaf's CUDA and OpenCL ports are aimed primarily at accelerator cards. The CUDA implementation specifically targets NVIDIA GPUs. There is also an implementation that uses OpenACC directives, to offload the computation to accelerator devices, including NVIDIA GPUs and Intel's Knights Corner. Each of these manual ports are standalone programs, replicating the full mini-app that has over 7000 LoC, and require maintenance by the authors of the code. The latest versions can be found on the UKMAC website and GitHub repository [17].

B. OPS

OPS (Oxford Parallel Library for Structured-mesh solvers) is a domain specific language embedded in C/C++ and Fortran [6]. It consists of a domain specific API that facilitates the development of applications operating over a multi-block structured mesh. Such a mesh can be viewed as an unstructured collection of structured mesh blocks, together with associated connectivity information between blocks. Using OPS, an application developer can write a multi-block structured-mesh application using the API as calls to a traditional library. A source-to-source translator is then used to parse the API calls and produce different parallelisations. A number of mini-apps have been re-engineered to use the OPS API, including CloverLeaf [20] and TeaLeaf.

Currently, OPS is able to automatically produce code that makes use of a range of parallel programming models and extensions such as OpenMP, CUDA, OpenCL, OpenACC and their combinations with MPI. The generated code attempts to use the best optimisations for the given programming model. Examples include the use of cache-blocking tiling to reduce data movement in the OpenMP and MPI versions of the generated code [21]. The key advantage of using OPS is that all these parallelisations and optimisations are produced automatically, from a single high-level source, without the need for maintaining each parallel version.

C. Kokkos and RAJA

Kokkos and RAJA are both C++ template libraries, designed with a similar goal to OPS. Through template metaprogramming, they aim to add portability to applications. They are also able to handle a wider range of domains.

Kokkos is able to select the most appropriate data layout (array of structures (AoS) or structure of arrays (SoA)) based

Version		Compiler	Flags
Manual	OpenMP MPI OpenMP and MPI	Intel 17.0u2, IMPI 2017u2	`-O3 -no-prec-div -fpp -align array64byte -qopenmp -ip -fp-model strict -fp-model source -prec-div -prec-sqrt`

	CUDA	Intel 17.0u2,
		CUDA 8.0.61
			`nvcc -gencode arch=compute_60,code=sm_60 -restrict -DNO_ERR_CHK -O3`
			`ifort -O3 -fpp -no-prec-div -qopenmp -fp-model strict -fp-model source -prec-div -prec-sqrt`
			`icc -O3 -qopenmp -fp-model strict -fp-model source -prec-div -prec-sqrt`

	OpenACC	PGI 17.3, Open-MPI 1.10.6	`-O3 -acc (-ta=multicore` **`or`** `-ta=tesla:cc60) -mp`
OPS	OpenMP MPI OpenMP and MPI MPI Tiled	Intel 17.0u2, IMPI 2017u2	`-O3 -ipo -fp-model strict -fp-model source -no-prec-div -prec-sqrt -vec-report2 -xHost -parallel -restrict -fno-alias -inline-forceinline -qopenmp`
	CUDA (OPS_BLOCK_SIZE_X=64, OPS_BLOCK_SIZE_Y=8)	Intel 17.0u2, IMPI 2017u2, CUDA 8.0.61	`nvcc -O3 -restrict --use_fast_math -gencode arch=compute_60,code=sm_60` `icc -O3 -ipo -fp-model strict -fp-model source -no-prec-div -prec-sqrt -vec-report2 -xHost -parallel -restrict -fno-alias -inline-forceinline -qopenmp`
	OpenACC	PGI 17.3, Open-MPI 1.10.6	`-acc -ta=tesla:cc60 -O2 -Kieee -Minline -ldl`
Kokkos	OpenMP	Intel 17.0u2	`-O3 -no-prec-div -fpp -fp-model strict -fp-model source -prec-div -prec-sqrt`
	CUDA	GNU-5.4.0, CUDA 8.0.61	`-O3 -march=native -funroll-loops -DKOKKOSP_ENABLE_PROFILING -ffloat-store`
RAJA	OpenMP	Intel 17.0u2, IMPI 2017u2	`-O3 -no-prec-div -restrict -fno-alias -xhost -std=c++11 -qopenmp -DNO_MPI -DENABLE_PROFILING`
	CUDA	GNU-5.4.0, CUDA 8.0.61	`nvcc -ccbin g++ -O2 --expt-extended-lambda -restrict -arch compute_60 -std=c++11 -Xcompiler -fopenmp --x cu` `icpc -march=native -funroll-loops -std=c++11 -ffloat-store -fopenmp`

TABLE I: List of all versions of TeaLeaf with compilers and corresponding flags used on the single node, multi-core systems

on the underlying architecture. It is able to produce three on-node/shared-memory parallelisations: PThreads, OpenMP and CUDA [7].

Similar to Kokkos, RAJA [8] is a template based C++ library that can be used to produce parallelised implementations of programs. RAJA also uses lambda functions in order to allow for more flexibility when building kernels. This model allows for parallelisation through OpenMP, CUDA and their combinations with MPI.

IV. Performance

Our aim is to compare each of the different implementations of TeaLeaf as described in Section III in terms of their efficiency across multiple different systems. By doing so, we can determine which frameworks perform better, and which multi-core/many-core systems are able to offer performance increases for TeaLeaf.

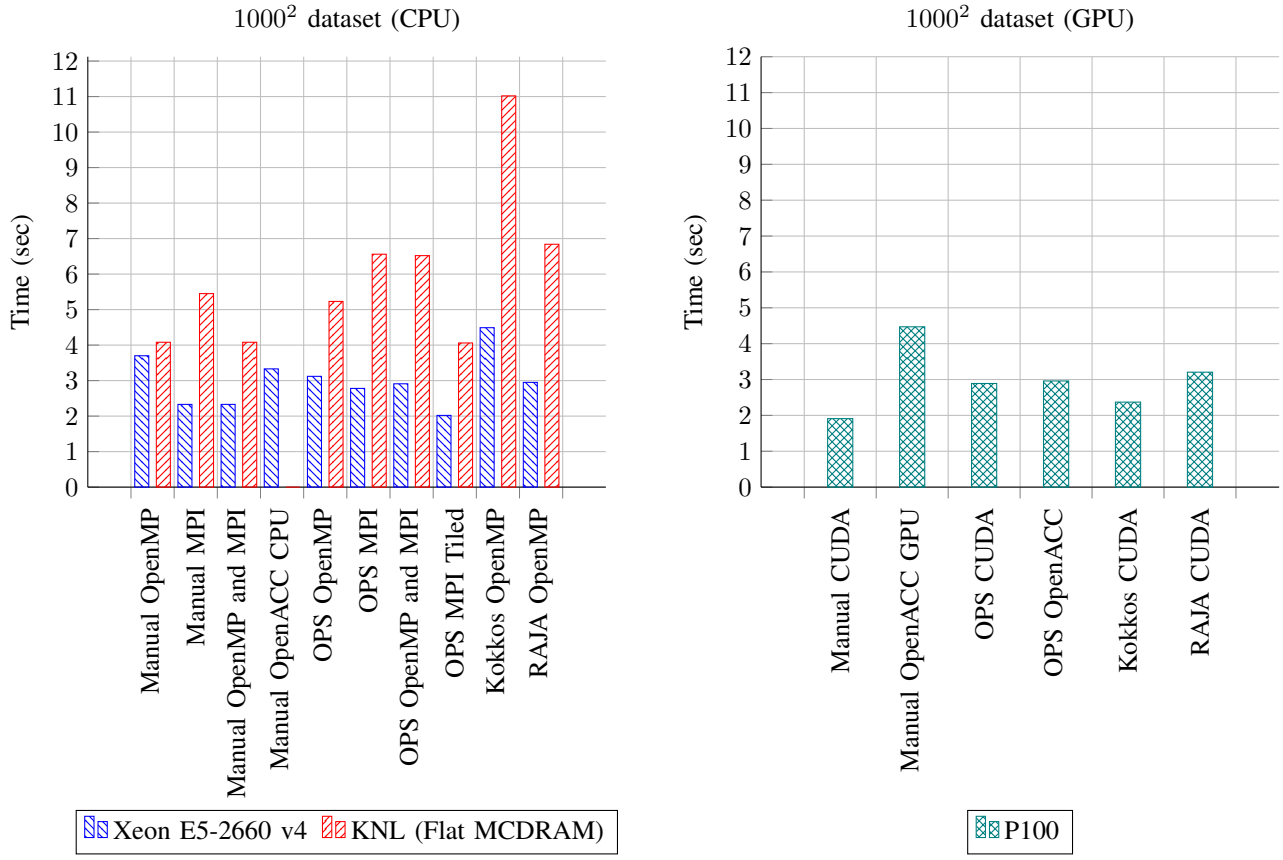
A. Experimental Setup

The results in this paper have been collected from 3 different, single node, multi-core/many-core systems. Each of these systems have been configured with the same set of compilers (which are described later), Linux kernel (3.16.0-4-amd64) and operating system (Debian GNU/Linux 8), in order to get comparable results. These systems can be found listed in Table II.

System	Key information
Intel Xeon E5-2660 v4	2 processors, each with 14 core and 2 hyperthreads per core. 2.00GHz
Intel Xeon Phi 7210 (KNL)	1 processor with 64 cores and 4 hyperthreads per core. 1.30GHz, Flat memory mode, Quadrant clustering mode
NVIDIA Tesla P100	3840 single precision CUDA cores (1920 double precision CUDA cores).

TABLE II: List of all single node, multi-core systems used to test different versions of TeaLeaf

The compilers and flags used on each of the implementations can be seen in Table I. Where possible, the Intel compiler (17.0u2) and Intel MPI (2017u2) were used when using Intel hardware. There were two exceptions to this: (1) when using the C++ template libraries Kokkos or RAJA with CUDA, GNU 5.4.0 and CUDA 8.0.61 were employed; and, (2) when using OpenACC, the PGI compiler (17.3) and OpenMPI (1.10.6) were used to enable support for OpenACC pragma statements.



(a) Times for TeaLeaf using 1000^2 dataset on CPU systems

(b) Times for TeaLeaf using 1000^2 dataset on GPU systems

Fig. 1: Performance of all implementations of TeaLeaf on all systems specified in Table II

For the Tesla P100 system, CUDA 8.0.61 was used.

Some of the versions, such as OPS's CUDA, can take parameters at runtime to further optimise the program. On this implementation, the block size for the kernels can be set by the user to allow for better performance on GPUs. For this paper, the block size has been set to $(64, 8)$ as this was shown to provide the largest improvements.

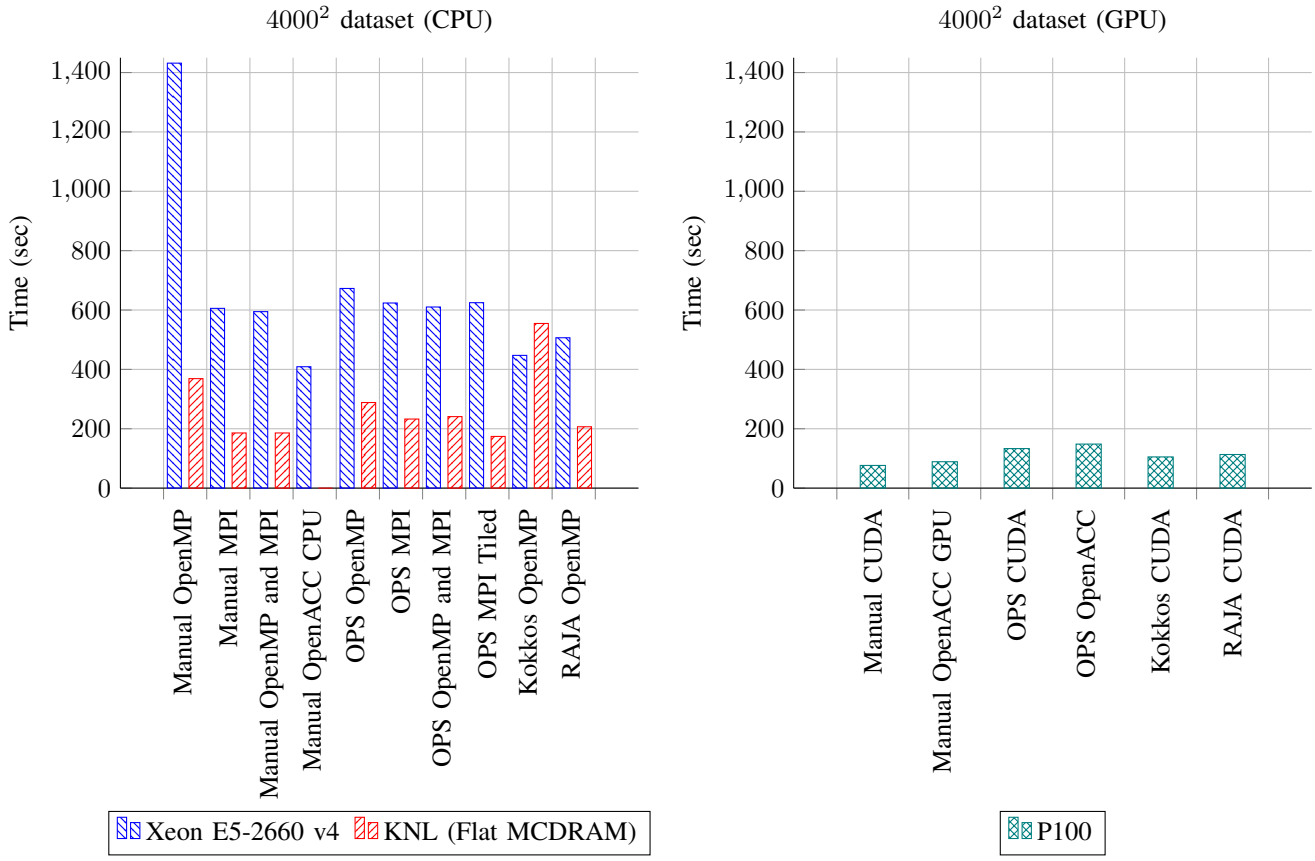
B. Results

Figures 1 and 2 detail the performance on each system. Figure 1 presents the time taken by ten iterations of the main time-marching loop of TeaLeaf solving a 2D problem size of 1000^2. Figure 2 shows the same but for the larger problem size of 4000^2. In Figures 1a and 2a, the first four sets of columns represent results from manually parallelised versions of TeaLeaf on the Xeon CPU and the Knights Landing system. The next four groups are from OPS on the same systems, and the final three groups represent the C++ template libraries Kokkos and RAJA. Figures 1b and 2b show the performance of implementations capable of running on GPU architectures. The first two bars represent the manually parallelised CUDA and OpenACC implementations, the third and fourth bars represent the OPS' CUDA and OpenACC versions, and the final two bars represent the Kokkos and RAJA CUDA implementations.

The times given in Figures 1 and 2 are the minimum execution times given all the available options for an implementation. For example, the OpenMP versions were tested over a large range of configurations to find the optimal number of threads. Of particular note, the high bandwidth memory (MCDRAM) for the Knights Landing system was set up to be in flat mode, using Quadrant clustering [22]. This allowed for the memory to be separately addressable and allocates the memory to the closest set of processors. Our experiments showed that this configuration provided the fastest run times compared to the other memory modes. To access this memory, *numactl* was used to allocate all the memory required by the program to the MCDRAM. Should the MCDRAM run out of available memory, *numactl* would start to use the available DDR memory.

OpenMP and MPI

The only parallelisation model used within all the libraries tested is OpenMP. This provides an opportunity to compare each of the libraries with a consistent model. OpenMP was the slowest on all systems using CPU architectures when using the small problem set. The slowest two executions were achieved by Kokkos, with a runtime of 4.49 seconds on the Xeon E5-2660 v4, and 11.02 seconds on the Knights Landing system. Out of all the OpenMP versions, the manual implementation of OpenMP on the KNL achieved close to the fastest time for the platform, with OPS's MPI Tiled implementation matching this performance or marginally performing better. This is not the case when looking at the larger dataset, where the manually parallelised version of OpenMP achieved the worst time out



(a) Times for TeaLeaf using 4000^2 dataset on CPU systems

(b) Times for TeaLeaf using 4000^2 dataset on GPU systems

Fig. 2: Performance of all implementations of TeaLeaf on all systems specified in Table II

of any implementation when run on the Xeon. However, this appears to be an outlier, being almost $3\times$ slower than any other implementation. Particularly, the manually parallelised version using MPI is almost always faster than its OpenMP counterpart. NUMA issues may be contributing to part of this performance degradation, but it is apparent that further optimisations may be required for the manual OpenMP version to improve performance. The best OpenMP performance on the KNL system for the larger dataset is given by the version using the RAJA library.

Most of the frameworks used to parallelise TeaLeaf include an MPI implementation. All of the MPI implementations tested also contain an option to use OpenMP alongside MPI. With MPI+OpenMP, TeaLeaf often performed better than the equivalent, OpenMP only implementation. OPS allows the user to generate code with specific optimisations on top of the MPI+OpenMP parallelisation. One such optimisation allows for cache-blocking tiling to reduce data movement [21]. The tiling optimisation made the code faster than the equivalent OpenMP and MPI+OpenMP implementations without tiling. This is especially true for the KNL system, where it gained the fastest time for the small dataset and the second fastest for the larger dataset.

RAJA and Kokkos

Out of all of the OpenMP implementations tested on the CPU architectures, RAJA gave the best performance on the the small dataset using the Xeon system, and the large problem size on the KNL. In contrast, the Kokkos implementation was often the slowest out of all OpenMP implementations, the exception being the large dataset being run on the KNL system.

While Kokkos' OpenMP implementation of TeaLeaf may not perform well on either the Xeon or the KNL, the CUDA version does perform better on NVIDIA's Tesla P100 GPU. For both problem sets, the Kokkos implementation was faster than the OPS and RAJA versions designed for GPUs. However, the fastest variant of TeaLeaf on a GPU is the manually parallelised implementation using CUDA.

For both the small and large problem sizes, RAJA's CUDA implementation is slower than both the manually implemented CUDA version and the Kokkos implementation. Using the larger dataset, RAJA CUDA was quicker than all of the OPS implementations. However, the same cannot be said for the smaller dataset, where it is slower than all implementations of OPS running on the P100 system.

OpenACC

Another parallelisation model that is predominately designed for GPU compilation is OpenACC. Two OpenACC implementations were tested on the P100 GPU, one generated using OPS and one which was manually implemented. For the larger problem set, the manually parallelised OpenACC implementation performs very well, getting the second fastest

time running on the Tesla P100. However, both OpenACC implementations are slower than the Kokkos CUDA implementation using the smaller dataset. When using both datasets, the CUDA implementations of TeaLeaf is faster than the OpenACC counterparts.

As well as offloading to the GPU, OpenACC can offload to the host processor. This means that the CPU can do all of the processing that would be executed on the GPU. Currently, OPS's OpenACC implementation does not support offload to the host device, so this was tested using the manually parallelised version of TeaLeaf OpenACC. For the smaller dataset, the OpenACC implementation on CPUs performs marginally better than the manually parallelised OpenMP and Kokkos versions. However, it is slower than both OPS's and RAJA's OpenMP implementations. On the larger problem size, the manually parallelised OpenACC version performs extremely well, performing the best of any implementation on the Xeon. OpenACC cannot offload to a KNL as a host device using the PGI 17.3 compilers, so could not be tested with the OpenACC implementation.

C. System Analysis

Between the two Intel architectures, performance on the Xeon was generally greater than the KNL when the smaller problem size was used. With the 1000^2 dataset, the application requires in the region of 200 MB of memory; for the 4000^2 dataset, this increases to 2.5 GB. Analysing the caching behaviour for the two cases shows that the Xeon system has a third of the cache misses of the KNL for the small dataset. For the larger dataset, the KNL has a less cache misses, and less cache accesses overall. The application is memory-bound (as we shall see later) and the MCDRAM therefore increases performance.

The P100 specific implementations are generally more performant than those that can be run on either the Xeon or KNL systems, when using the large problem set. However, the percentage difference between the fastest time on a GPU compared to the fastest on a CPU is not as large when the smaller dataset is used (3.04% for the small dataset, 50.57% for the larger dataset). This is an expected performance trait of GPUs where smaller problem sizes benefit less from the increased parallelism available. Overheads (as a proportion of total run time), such as kernel calls and memory copies, further reduce performance when working on smaller problem sizes.

V. Performance Portability

Performance portability has been a topic of interest within HPC community for some time; the US Department of Energy's Centers of Excellence Performance Portability Meeting was set up specifically to discuss how to mitigate the problems with platform diversification and how different laboratories are working on the issue. During and following the April 2016 meeting, an attempt was made to establish a more concrete definition of performance portability. *Performance* and *Portability* are subjective terms, heavily dependent on the user's point of view and the problem being solved [23]. One similarity in all definitions was the intuition that a performance portable code should be able to run on a variety of machines. There have been many different approaches to solve this, including compiler directives such as OpenACC [24] and OpenMP, languages designed for performance portability such as Chapel [25] and PetaBricks [26], execution models such as EARTH [27], and using embedded domain specific languages such as OPS [6] and OP2 [28]. Template libraries have also been used to add performance portability to an application, examples of which include Kokkos [7] and RAJA [8].

Assessing the portability of a particular program is usually done by measuring performance on multiple machines and then comparing the results. Quantifying how "performance portable" an application is from these results is difficult. To remedy this, Pennycook et al. [9] propose the metric:

$$P(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where, H is the set of systems used to test the application, and e is the efficiency of the application a given the input parameters p [9]. The metric uses the harmonic mean to assess either: (i) the application efficiency, i.e., how fast the application runs compared to the best time on each system; or, (ii) the architecture efficiency, i.e., the achieved number of floating point operations per second compared to the maximum possible on each system. The resulting score ranges between 0% and 100%; should the program not be portable to one or more systems, a score of 0% is achieved.

In this paper, we use the metric to evaluate the different versions of TeaLeaf. Because the systems tested fall under two distinct architectures: CPUs and GPUs, two sets of performance measures have been taken. The first looks at the CPU architectures only and the second looks at all available systems. This means that some of the implementations of TeaLeaf can be compared to the other implementations even though an implementation cannot be run on a particular system.

Table III shows the performance portability of different versions of TeaLeaf using the larger dataset (4000^2). In order to compare them all effectively, the manually parallelised implementations have been combined together into one version, named "Manual". The best performing implementation running the best options was then used for the architecture and the application efficiency. Note that the implementation that achieves the best architecture efficiency may not also achieve the best application efficiency and vice versa. In order to effectively represent the architecture efficiency, we calculated two metrics. The first is the achieved number of FLOPs (i.e. compute intensity) for each parallelisation and the second is the memory bandwidth used. Both measures were obtained using, Intel's *VTune 2017* profiler for the CPU systems, and NVIDIA's CUDA profiler *nvprof* for the GPU systems.

Table III has been laid out such that the efficiencies for the two CPU architectures are given, followed by the performance portability of these architectures. The efficiency of the frame-

Version	Eff. (Xeon) (%)			Eff. (KNL) (%)			P(CPU) (%)			Eff. (P100) (%)			P(CPU $\cup$ GPU) (%)		
	Arch.		App.	Arch.		App.	Arch.		App.	Arch.		App.	Arch.		App.
	Com.	BW		Com.	BW		Com.	BW		Com.	BW		Com.	BW	
Manual	0.96	60.49	100.00	1.52	91.61	93.73	1.18	73.19	96.76	2.36	75.70	100.00	1.42	74.01	97.82
OPS	1.35	89.61	67.02	3.39	95.93	100.00	1.93	92.66	80.26	2.83	61.21	57.32	2.16	79.11	70.81
Kokkos	2.73	64.11	91.45	1.57	23.59	31.40	2.00	34.49	46.74	5.30	65.86	72.65	2.52	41.00	53.05
RAJA	0.91	53.13	80.73	1.60	60.87	84.25	1.16	56.74	82.45	1.87	70.63	67.46	1.33	60.72	76.77

TABLE III: Performance Portability on Xeon E5-2660 v4, KNL (MCDRAM) and a P100 card for the 4000^2 mesh

works on the P100 system are then presented, accompanied by the performance portability of all three systems. For completeness, the architecture efficiency has been split into two sections, one for the compute performance and one for the memory bandwidth.

A. Architecture Efficiency

From Table III, we can see that the compute efficiency is a significantly smaller portion of the system peak, on all systems. Barely 5% of the peak is achieved. However the bandwidth efficiency is mostly over 50%. As such, it is clear that TeaLeaf is a bandwidth limited application. Therefore, we will concentrate only on the architectural efficiency related to BW, in this section.

With the exception of Kokkos on the KNL, the amount of bandwidth used by the different parallelisation models exceeds 60%. The highest bandwidth usage was achieved by OPS on the KNL, utilising 95.93% of the available bandwidth. When looking specifically at the KNL results, the amount of bandwidth used correlates with the application efficiency, with models using more bandwidth gaining the higher application efficiency. This is to be expected, as we would expect a faster program to better utilise the hardware available. Over all the CPU architectures, OPS achieved the highest bandwidth, and thus gained the largest performance portability for CPU systems.

Looking at the Tesla P100 system, we can see that the bandwidth efficiency is relatively high, and spread over a small range (14.49% difference). As with the KNL system, the fastest implementation got the highest bandwidth. However, unlike the KNL system, the highest bandwidth utilisation was achieved by the manually parallelised implementation. This leads to both the manually parallelised and OPS versions having very close performance portability based on the architecture efficiency (74.01% and 79.11% respectively).

B. Application Efficiency

Delving into the application efficiency, nearly all the results on the CPU architectures are greater than 80%. The exceptions are OPS on the Xeon (67.02%), and Kokkos on the KNL (31.40%). These low results are reflected in the performance portability metric for the CPU, where Kokkos is approximately 34% away from the the next highest performance portability score across all CPU architectures.

As stated previously, almost all the other implementations of TeaLeaf performed very well, getting above 80% efficiency. This is reflected in the performance portability metric, with the highest being 96.76% by using the manual implementation. Both OPS and RAJA achieved very similar performance portability scores across both CPU architectures, with only a 2.19% difference.

However, very few implementations gained a high application efficiency when executed on the P100 system. The manually parallelised versions were the fastest, with Kokkos coming in second with a 72.65% application efficiency.

Due to the low performance portability on the CPU architectures, Kokkos' overall performance portability for application efficiency was the lowest out of all the frameworks, scoring 53.05%. On the other hand, the manually parallelised implementations scored the highest out of all models, being the only one to score above 90%. This very much agrees with the intuition that manually optimising and parallelising the code will get the best results, even if this means longer development time. Both OPS and RAJA got lower performance portability once the GPU architecture was included.

VI. Conclusion and Future Work

In this paper, we investigated the performance of different implementations of TeaLeaf, a mini-application that solves the linear heat conduction equation. First, we looked at the performance of the mini-app across 3 different multi-core systems: Intel's Xeon E5-2660 v4 CPU; Intel's Xeon Phi Knights Landing processor; and, NVIDIA's Tesla P100 GPU. We showed that the GPU implementations of the different frameworks were faster for larger datasets, with the KNL system closely behind. The best times on the CPU were achieved by the manually parallelised OpenACC implementation and the MPI tiled implementation of OPS.

Secondly, we looked at the *performance portability* of different version of TeaLeaf. Overall, the architecture efficiency based on compute intensity (FLOPs/s) was significantly low. However, this was expected, as real-world programs such as TeaLeaf, are usually more complex than traditional benchmarking applications such as LINPACK, that typically designed to stress the hardware fully. One the other hand, architecture efficiency based on the bandwidth was almost always over 50%, leading us to conclude that TeaLeaf is a memory bound application.

OPS's architectural efficiency, based on bandwidth, was the highest on CPU architectures. However, for the GPU systems, the manually parallelised version utilised a higher percentage of the peak bandwidth. Overall, both OPS and manual implementations achieved comparable architecture efficiencies.

In terms of application efficiency, the manually parallelised implementations achieved the highest scores, showing that

hand-coding the parallelisations and optimisations will typically produce better results. However, the downside to this method is the need to develop and maintain each separate version. Out of all the library based methods, both OPS and RAJA produced good performance. Both OPS and RAJA achieved above 70% overall performance portability.

A. Future Work

Future work for this research will include further investigating the performance of TeaLeaf on heterogeneous architectures, specifically with regards to memory and cache usage. We also aim to examine the difference between single node and distributed memory systems containing the same multi-core processors, and investigate how performance portability will change for codes developed with each of the frameworks.

ACKNOWLEDGEMENTS

This work was supported by the UK Atomic Weapons Establishment under grant CDK0724 (AWE Technical Outreach Programme). Prof. Stephen Jarvis is an AWE William Penney Fellow. This work would not have been possible without the assistance of a number of members of the Applied Computer Science group at AWE, to whom we would like to express our gratitude.

The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications" project. This research was also funded by the Hungarian Human Resources Development Operational Programme (EFOP-3.6.2-16-2017-00013) and by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences.

Additionally, we are grateful for the support of Dr. John Pennycook at Intel for his comments and guidance.

REFERENCES

[1] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[3] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[4] UKMAC, "UK Mini-App Consortium (UKMAC)," http://uk-mac.github.io/, Accessed: 2017-04-25.

[5] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the Performance Portability of Modern Parallel Programming Models Using TeaLeaf," *Concurrency and Computation: Practice and Experience*, 2017.

[6] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Design and Development of Domain Specific Active Libraries with Proxy Applications," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 738–745.

[7] H. C. Edwards and C. R. Trott, "Kokkos: Enabling Performance Portability Across Manycore Architectures," in *Extreme Scaling Workshop (XSW'13)*. Boulder, CO: IEEE Computer Society, Los Alamitos, CA, August 2013, pp. 18–24.

[8] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. LLNL-TR–661403, 2014.

[9] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," *arXiv preprint arXiv:1611.07409*, 2016.

[10] D. A. Nowak and R. C. Christensen, "ASCI Applications," Lawrence Livermore National Lab., CA (United States), Tech. Rep., 1997.

[11] K. Franko, "MiniAero and Aero: An Overview," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2015-1456PE, 2015.

[12] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly, "Performance Analysis of the OP2 Framework on Many-Core Architectures," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 9–15, March 2011.

[13] T. Deakin, S. McIntosh-Smith, and W. P. Gaudin, "Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale," in *High Performance Computing*, D. J. Kunkel J., Balaji P., Ed., vol. 9697. Springer-Verlag, Berlin, June 2016, pp. 429–448.

[14] UKMAC, "BookLeaf," https://github.com/UK-MAC/BookLeaf, Accessed: 2017-06-28.

[15] ——, "CloverLeaf," https://github.com/UK-MAC/CloverLeaf, Accessed: 2017-06-28.

[16] S. R. Sukumar, M. A. Matheson, R. Kannan, and S.-H. Lim, "Mini-apps for High Performance Data Analysis," in *IEEE International Conference on Big Data*. IEEE, 2016, pp. 1483–1492.

[17] UKMAC, "TeaLeaf," http://uk-mac.github.io/TeaLeaf/, Accessed: 2017-04-25.

[18] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. P. Gaudin, "An Evaluation of Emerging Many-core Parallel Programming Models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2016, pp. 1–10.

[19] M. Martineau, S. McIntosh-Smith, M. Boulton, W. P. Gaudin, and D. Beckingsale, "A Performance Evaluation of Kokkos & RAJA using the TeaLeaf Mini-App," in *Supercomputing 2015 (SC15)*, Austin, TX, November 2015.

[20] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman, "Performance Analysis of a High-Level Abstractions-Based Hydrocode on Future Computing Systems," in *Proceedings of the Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2014, pp. 85–104.

[21] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop Tiling in Large-Scale Stencil Codes at Run-time with OPS," *CoRR*, vol. abs/1704.00693, 2017. [Online]. Available: \url{http://arxiv.org/abs/1704.00693}

[22] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[23] J. R. Neely, "DOE Centers of Excellence Performance Portability Meeting," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2016.

[24] S. Sawadsitang, J. Lin, S. See, F. Bodin, and S. Matsuoka, "Understanding Performance Portability of OpenACC for Supercomputers," in *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015, pp. 699–707.

[25] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua, "Performance Portability with the Chapel Language," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 582–594.

[26] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable Performance on Heterogeneous Architectures," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 431–444.

[27] W. Zhu, Y. Niu, and G. R. Gao, "Performance Portability on EARTH: A Case Study Across Several Parallel Architectures," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2005, p. 8.

[28] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford, "Acceleration of a Full-scale Industrial CFD Application with OP2," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016.